LLMalMorph：论使用大型语言模型生成恶意软件变体的可行性

摘要

Large Language Models (LLMs) have transformed software development and automated code generation. Motivated by these advancements, this paper explores the feasibility of LLMs in modifying malware source code to generate variants. We introduce LLMalMorph, a semi-automated framework that leverages semantical and syntactical code comprehension by LLMs to generate new malware variants. LLMalMorph extracts function level information from the malware source code and employs custom-engineered prompts coupled with strategically defined code transformations to guide the LLM in generating variants without resource-intensive fine-tuning. To evaluate LLMalMorph, we collected 10 diverse Windows malware samples of varying types, complexity and functionality and generated 618 variants. Our thorough experiments demonstrate that it is possible to reduce the detection rates of antivirus engines of these malware variants to some extent while preserving malware functionalities. In addition, despite not optimizing against any Machine Learning(ML)-based malware detectors, several variants also achieved notable attack success rates against an ML-based malware classifier. We also discuss the limitations of current LLM capabilities in generating malware variants from source code and assess where this emerging technology stands in the broader context of malware variant generation.

关键词：

Abstract

大型语言模型（LLMs）已经改变了软件开发和自动化代码生成。受这些进展的激励，本文探索了 LLMs 修改恶意软件源代码以生成变种的可行性。我们引入了 LLMalMorph，这是一个半自动化框架，它利用 LLMs 对代码的语义和语法理解来生成新的恶意软件变种。LLMalMorph 从恶意软件源代码中提取函数级信息，并采用定制设计的提示词结合策略性定义的代码转换，来指导 LLM 生成变种，而无需资源密集型的微调。为了评估 LLMalMorph，我们收集了 10 个不同类型、复杂度和功能的多样化 Windows 恶意软件样本，并生成了 618 个变种。我们详尽的实验表明，在保持恶意软件功能的同时，可以在一定程度上降低这些恶意软件变种对防病毒引擎的检测率。此外，尽管没有针对任何基于机器学习（ML）的恶意软件检测器进行优化，一些变种在对抗一个基于 ML 的恶意软件分类器时也取得了显著的攻击成功率。我们还讨论了当前 LLM 在从源代码生成恶意软件变种方面的能力局限性，并评估了这项新兴技术在更广泛的恶意软件变种生成背景下的现状。

Key Words:

目录

第 1 章 引言 1
1.1 先前的研究 1
1.2 问题描述 2
1.3 我们的方法 2
1.4 实验和分析 3
1.5 贡献 3
1.6 开源 4
第 2 章 背景 5
2.1 恶意软件和检测手段 5
2.1.1 LLMs 和提示词工程 5
第 3 章 概述 7
3.1 A. 问题描述 7
3.2 B. 挑战与解决方案概述 7
3.2.1 (C1) 编辑恶意软件源码中的上下文与结构挑战 7
3.2.2 (A1) 在功能级别生成恶意软件变体的框架 8
3.2.3 (C2) 使用基于 LLM 的函数修改时恶意软件代码库一致性保持的挑战 8
3.2.4 (A2) 融入人在回路流程 8
第 4 章 LLMalMorph 的详细设计 9
4.1 A. LLMalMorph 框架 9
4.1.1 Extractor 子模块 10
4.1.2 Prompt Generator 子模块 10
4.1.3 LLM Based Function Modifier 子模块 10
4.1.4 Merger 子模块 11
4.1.5 Compilation and Debugging 子模块 11

4.2 B. 代码转换策略 12
4.2.1 代码优化 12
4.2.2 代码质量与可靠性 12
4.2.3 代码可重用性 12
4.2.4 代码安全性 12
4.2.5 代码混淆 12
4.2.6 Windows API 特定转换 13
4.3 C. LLM 的快速设计 13
第 5 章 评估 15
5.1 评估设置 15
5.1.1 选择样本 15
5.1.2 评估指标 15
5.1.3 策略导向的机器学习分类器攻击成功率 (ASR) 16
5.1.4 策略导向的代码编辑工作量 (W_s^M) 16
5.1.5 人力投入量化指标 (H_s^M) 16
5.1.6 功能保留指标 (Φ^M) 16
5.2 模型选择 17
5.3 实现细节 17
5.4 评估结果与分析 18
5.4.1 对研究问题 1（RQ1）的解答 18
5.4.2 VirusTotal 18
5.4.3 混合检测 20
5.4.4 机器学习分类器 21
5.4.5 比较分析 21
5.4.6 研究问题 2（RQ2）的解答 23
5.4.7 代码编辑工作量 23
5.4.8 人力投入 24
5.4.9 对研究问题三（RQ3）的解答 24

第 6 章　相关工作 27
第 7 章　结论和未来工作 28
第 8 章　致谢 29
第 9 章　附录 A 30
参考文献 31
攻读学位期间发表论文与研究成果清单 32

插图

图 4.1　LLMalMorph 整体架构。该框架由两大核心模块构成：功能变异模块：从恶意软件源代码文件中提取功能函数，并借助 LLM 进行修改。变种合成模块：将修改后的函数更新至恶意软件源代码，通过编译项目生成变种文件 9

表格

表 1.1　与先前研究的对比 2
表 4.1　选择的恶意软件样本总结 14
表 5.1　各策略的 ASR(%)。百分比已对不同变体数量进行标准化（Fungus：9 个变体/策略；Dexter：12 个；Conti：14 个；Babuk：11 个）。 21
表 5.2　VirusTotal 和混合分析对于 Malguise 和 LLMalMorph 生成的对抗变种 AV 检测率（%）的比较。 22
表 5.3　所有样本在 2 个反病毒检测器下的功能保留率 Φ^M 25
表 9.1　修改的函数选择标准 30
表 9.2　每个恶意软件样本的函数选择 30

主要符号对照表

LLM	大语言模型的英文缩写

第 1 章 引言

恶意软件（Malware）继续随着技术的快速扩张而激增。到 2025 年，网络犯罪造成的损失预计将达到每年 10.5 万亿美元 [1]。每秒大约发生 19 万起新的恶意软件事件 [2]，而 2024 年勒索软件的平均赎金要求预计将达到每次攻击 273 万美元，较往年急剧上升 [3]。尽管经过数十年的研究和缓解努力，这些数字突显了恶意软件研究在当今不断演变的威胁环境中的紧迫重要性。

现代最具变革性的 AI 技术之一是大型语言模型（LLMs），它在自然语言处理（NLP）[4]–[6]、代码生成 [7]–[12] 以及代码编辑和重构等软件工程任务 [13]–[15] 中展现了非凡的能力。鉴于这些优势和进步，利用 LLMs 进行恶意软件源代码转换是自然的发展。最近一项针对全球行业 1800 名安全负责人的调查 [16] 发现，74% 的人正经历着显著的 AI 驱动的威胁，60% 的人感觉准备不足，无法抵御这些威胁。尽管当前的模型仅从文本生成功能完整的恶意软件存在显著局限性，但研究表明它们可以生成恶意行为者能够组装成可操作恶意软件的代码片段 [17]。LLM 能力的进步与恶意软件威胁的演变相结合，为对手使用这些模型创建新恶意软件并将现有代码库变异成更难以捉摸和更具破坏性的变种铺平了道路。尽管恶意软件源代码比二进制文件更难获取，但能够访问源代码的对手，例如恶意软件作者、泄露存储库的用户或修改开源恶意软件的人，仍然可以利用 LLMs 生成新的、更难检测的变种。这些模型使攻击者能够持续精进和扩展其武器库，从而大规模增加恶意活动的持久性和规避性。

1.1 先前的研究

先前的研究提出了各种创建恶意软件变种的方法 [17]–[23]。然而，这些方法在至少以下一个方面表现出局限性（如表1.1所示）(A) 大多数现有方法没有利用 LLMs 来转换恶意软件的源代码 [18]–[23]；(B) 大多数方法依赖迭代算法来生成恶意软件变种 [18]–[20], [22], [23]；(C) 使用 LLMs 进行变种生成的方法，直接从成功率低的提示词开始 [17]。此外，尚不清楚生成的恶意软件在规避广泛使用的防病毒引擎方面是否表现更优。鉴于目前的情况，我们的工作引入了一种与现有恶意软件变种生成方法截然不同的方法。与大多数先前主要依赖基于对抗性机器学习或基于搜索的方法的研究不同，我们的方法独特地利用 LLMs 在源代码级别进行操作。基本上，它从恶意软件源

代码开始，以高成功率和最少的手动工作生成变种。此外，我们的方法不需要迭代训练或基于搜索的优化，这使其与现有的恶意软件转换方法根本不同。因此，我们提出了一个尚未充分探索的新研究方向。

表 1.1　与先前研究的对比

方法	源代码	LLM 使用	无需训练或迭代	逃逸提升
Qiao, Yanchen, et al. [18]	否	否	否	是
Tarallo [23]	否	否	否	是
Malware Makeover [20]	否	否	否	是
MalGuise [22]	否	否	否	是
Ming, Jiang, et al. [21]	否	否	是	是
AMVG [19]	是	否	否	是
Botacin et al. [17]	否	是	是	否
LLMalMorph	是	是	是	是

1.2 问题描述

鉴于现有方法的局限性以及 LLMs（特别是代码生成方面）的最新进展，我们旨在回答以下问题——我们能否利用预训练 LLMs 的生成能力，无需额外微调，来开发一个半自动化且高效的框架，以生成保留功能语义的恶意软件变种，这些变种能够规避广泛使用的防病毒引擎和机器学习分类器？

1.3 我们的方法

在本文中，我们对上述问题给出了肯定的回答。我们设计、实现并评估了 LLMalMorph——一个专门用于生成用 C/C++ 编写的 Windows 恶意软件功能变种的框架。我们只专注于 Windows 恶意软件，因为它在消费者和企业环境中广泛使用，仍然是恶意软件最常针对的操作系统 [24], [25]。

LLMalMorph 结合了自动化代码转换和人工监督来生成恶意软件变种。该框架利用一个开源的 LLM，应用精心设计的转换策略和提示词工程，在保持结构和功能完整性的同时，高效地修改恶意软件组件。人机协同（human-in-the-loop）过程处理复杂转换和多文件恶意软件中的错误，允许进行调试和配置调整。这种半自动化方法也

使我们能够量化基于 LLM 从源代码生成恶意软件变种中的人力投入。

1.4 实验和分析

我们选择了 10 个不同复杂度的恶意软件样本，使用 6 种代码转换策略结合一个 LLM 生成了 618 个变种。我们使用主要依赖基于签名的检测和静态分析的引擎的 VirusTotal[1]和 Hybrid Analysis[2]评估了防病毒（AV）检测率，并测试了语义保留性。代码优化（Code Optimization）策略在两种工具上均持续实现了较低的检测率。平均而言，相对于每个样本的基准检测率，LLMalMorph 在 VirusTotal 上将简单样本的检测率降低了 31%，将三个更复杂样本的检测率降低了 10% 至 15%；在 Hybrid Analysis 上，与各自基准相比，四个样本的检测率降低了 8% 至 13%。除了 AV 工具外，我们还在一个基于机器学习（ML）的恶意软件分类器上评估了 LLMalMorph，并观察到在特定样本上，优化（Optimization）策略和安全（Security）策略取得了较高的攻击成功率（分别高达 89% 和 91%）。诸如优化、安全和 Windows API 修改等策略需要更多手动编辑，其中 Windows 和安全策略需要更高的调试投入。值得注意的是，四个样本中超过 66% 的规避型变种保留了其语义，这证明了 LLMalMorph 生成功能规避型恶意软件的能力。

1.5 贡献

总结而言，我们有以下贡献：

• 我们设计并实现了 LLMalMorph，一个实用的 Windows 恶意软件变种生成框架，它使用一个开源的 LLM 和基于提示词（prompt-based）的代码转换。

• 我们在 LLMalMorph 中设计了一个人机协同（human-in-the-loop）机制，以解决 LLM 在调试多文件恶意软件源代码和项目级配置方面的局限性。

• 我们进行了广泛的实验，从 10 个样本生成了 618 个恶意软件变种，并评估了它们在 VirusTotal 和 Hybrid Analysis 上的检测率和语义保留性，以及在一个机器学习分类器（ML Classifier）上的攻击成功率。

• 我们使用代码编辑工作量（code editing workload）比较了不同代码转换策略的有效性，并讨论了 LLM 所犯错误的类型。

[1]https://www.VirusTotal.com/gui/home

[2]https://hybrid-analysis.com/

1.6 开源

LLMalMorph 框架及其所有相关组件可在 Github[3]找到。

[3]https://github.com/AJAkil/LLMalMorph

第 2 章 背景

本节描述与恶意软件（malware）、其检测系统以及大型语言模型（LLMs）相关的各种预备知识。

2.1 恶意软件和检测手段

恶意软件（Malware）指的是对手或攻击者用来在用户不知情的情况下，未经授权访问数字设备以破坏或窃取敏感信息的恶意程序 [26]。它是一个统称（umbrella term），用于描述广泛的威胁，包括木马（Trojans）、后门（backdoors）、病毒（viruses）、勒索软件（ransomware）、间谍软件（spyware）和僵尸程序（bots）[27]，针对多种操作系统，如 Windows、macOS、Linux 和 Android，以及各种文件格式，如可移植可执行文件（Portable Executable, PE）、MachO、ELF、APK 和 PDF [28]。在入侵系统后，恶意软件可以执行各种恶意活动，例如渗透网络、加密数据以勒索赎金或降低系统性能。

检测引擎和工具采用各种方法和工具来检测恶意软件。它们可以大致分为静态（static）、动态（dynamic）和混合（hybrid）方法 [28], [29]。静态检测在不执行恶意软件的情况下对其进行分析，依赖于诸如 PE 头信息（PE header information）、可读字符串（readable strings）和字节序列（byte sequences）等特征 [28]。动态检测涉及在受控环境（例如沙箱，sandboxes）中执行恶意软件，以监视运行时行为，如注册表修改（registry modifications）、进程创建（process creation）和网络活动（network activity）[28], [29]。混合检测结合了静态和动态特征，使用诸如操作码（opcodes）、对系统的 API 调用（API calls to the system）和控制流图（control flow graphs, CFGs）等数据 [28]。此外，基于启发式的检测（heuristic-based detection）使用启发式规则（heuristic rules）静态分析代码并动态分析行为，以确定恶意性 [30]。

2.1.1 LLMs 和提示词工程

LLMs 通过在翻译、摘要等任务中的卓越表现，改变了自然语言处理（NLP）的格局。基于 transformer 架构 [31]，LLMs 利用了自注意力机制（self-attention mechanisms）。它们以自监督（self-supervised）方式在大规模语料库上进行预训练，以形成对语料库的深度上下文理解。预训练后，这些模型经过微调（fine-tuned）或指令微调（instruction-tuned）以执行特定任务。

LLMs 在编程任务中也展现了显著的能力，一些专门模型在大量代码和自然语言指令上进行了训练 [8], [9], [11], [12], [14]。这些模型最突出的特性之一是在推理过程中无需任务特定微调即可生成零样本代码（zero-shot code）（无需显式示例或参考）。这是通过提示词工程（prompt engineering）实现的，其中精心设计的输入提示词（prompts）指导模型生成期望的输出 [32]，使其成为代码合成（code synthesis）和重构（refactoring）等编程活动的多功能工具。

第 3 章 概述

在本节中，我们正式定义我们的问题，并阐述这些挑战及相应的解决方案。

3.1 A. 问题描述

令 M 表示一个由 F 个文件组成的恶意软件程序，其中第 i 个文件 (1≤i≤F) 包含 G 个函数，记作 $\{f_1^i, f_2^i, ..., f_G^i\}$。对于由语言模型（LLM）应用的给定转换策略 s，我们的目标是生成一个恶意软件变种 M_s，其中第 i 个文件包含使用策略 s 生成的修改后函数 $\{\hat{f}_1^i, \hat{f}_2^i, ..., \hat{f}_j^i\}$，同时保留未修改的函数 $\{f_{j+1}^i, ..., f_G^i\}$。该过程首先涉及从第 i 个文件中提取第 j 个函数 f_j^i，并构建一个提示符 $p_s||f_j^i$，该提示符包含转换策略 s、提取的函数 f_j^i 以及相关上下文信息（如全局变量和头文件）。然后我们得到转换后的函数 $\hat{f}_j^i = LLM(p_s||f_j^i)$ 。随后，将修改后的函数 f_j^i 合并回源代码文件 i，生成一个修改后的文件，其中函数 $\{\hat{f}_2^i, ..., \hat{f}_j^i\}$ 被修改，而其余函数 $\{f_{j+1}^i, ..., f_G^i\}$ 保持不变。最后，重构的文件被编译以生成变种恶意软件 $\hat{M}_s$。

3.2 B. 挑战与解决方案概述

我们现在讨论指导我们设计 LLMalMorph 的主要挑战和解决方法。

3.2.1 (C1) 编辑恶意软件源码中的上下文与结构挑战

由于在训练语料库中包含了大量开源代码库 [8]–[10], [12]，LLM 研究的最新进展极大地改进了跨多种语言的代码生成。除了生成代码，使用 LLM 进行代码编辑和重构也正受到关注 [13], [14]。一个关键挑战是上下文限制：提供完整的源代码和转换指令通常会超出模型的输入能力并阻碍指令遵循，特别是对于较小的模型。此外，恶意软件代码中的功能通常分布在多个文件中，这进一步使编辑过程复杂化。在处理 C 和 C++ 时，这个问题变得更加明显，因为它们经常导致产生无法编译或无法按预期执行的错误代码 [33]。这与恶意软件高度相关，因为这些程序经常利用系统调用 API 进行注册表修改、执行网络系统调用、进程修改或实施反规避技术。在基于 Windows 的恶意软件中，这些操作严重依赖于 Windows API 调用结合 C/C++ 或 C# 功能。因此，鉴于原生系统 API 的复杂性和上下文限制，使用 LLM 编辑大规模的恶意软件源代码仍然是一个重大挑战。

3.2.2 (A1) 在功能级别生成恶意软件变体的框架

为了应对挑战 C1，LLMalMorph 通过几个关键阶段运作。它首先遍历恶意软件源代码文件的抽象语法树 (AST)，以系统地提取函数主体、头文件信息和全局变量声明。随后，提取出的组件作为开源 LLM 的输入，其中精心设计的提示指导函数修改过程。最后，修改后的组件被重新整合回源代码，生成原始恶意软件的功能性变体。这种方法确保了恶意软件组件的精确提取和修改，同时在整个转换过程中保持其结构完整性，且不会使 LLM 负担过重。

3.2.3 (C2) 使用基于 LLM 的函数修改时恶意软件代码库一致性保持的挑战

恶意软件项目通常跨越多个文件，其中修改一个部分通常需要跨其他相关文件进行协调更改。鉴于当前 LLM 的能力，在保持多文件代码库一致性的同时编辑源代码，在没有人工监督的情况下可能过于容易出错，因为 LLM 在多文件修改、依赖关系解析、项目级配置以及跨大型代码库的编辑方面存在困难 [34], [35]。例如，使用指定转换重构单个函数可能需要在关联的头文件中进行更新、在整个代码库中重构和重命名其用法、或添加新头文件、链接静态库、修改编译器指令、或更改整个项目的语言配置以适应 LLM 生成的代码。虽然添加头文件或在单个文件内重命名等简单任务可以实现自动化，但更复杂的多步骤修改在很大程度上取决于 LLM 生成更改的性质和特定恶意软件项目的结构，这使得一刀切的解决方案不可行。尽管像 Copilot3 这样底层使用 LLM 的技术改进了多文件处理，但上下文限制仍然是阻碍将开源 LLM 接入整个代码库的关键障碍，该代码库能够可靠地重构互连的代码库，因此在泛化到各种恶意软件项目方面存在不足。代码生成的 LLM 幻觉问题 [36], [37] 加剧了这些限制，导致新的复杂问题，如在代码生成过程中使用虚构的函数或误用现有 API。因此，调试 LLM 通常涉及试错，且其当前能力不足以处理超出简单语法和逻辑问题的复杂错误修复。

3.2.4 (A2) 融入人在回路流程

为了解决 C2，我们选择了一种部分自动化的解决方案来生成恶意软件功能变体。如 A1 所述，我们以自动化方式生成具有功能转换的源代码。然而，为了保持一致性和正确性，我们采用人在回路流程来处理跨多文件恶意软件项目的复杂调试和配置更改。

第 4 章 LLMalMorph 的详细设计

4.1 A. LLMalMorph 框架

在本节中，我们将详细阐述我们框架的架构（见图4.1）。LLMalMorph 分为两个主要模块。第一个模块，功能变异模块使用 LLM 和策略性生成的提示来转换恶意软件源代码函数。第二个模块，变种合成模块将转换后的函数集成回源代码，编译修改后的项目以生成恶意软件变体。该模块还融入了人在回路流程用于编译期间的调试。第一个模块又包含三个关键子模块：Extractor、Prompt Generator 和 LLM Based Function Modifier。第二个模块包含两个主要子模块：Merger 以及 Compilation and Debugging。我们现在介绍支撑该框架的形式化算法，随后对各模块进行详细解释。

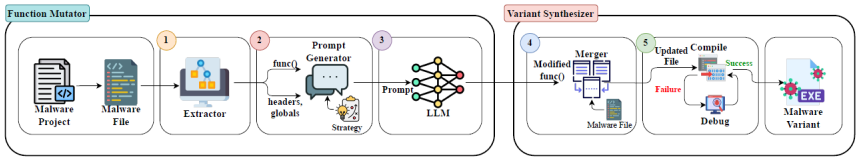

图 4.1 LLMalMorph 整体架构。该框架由两大核心模块构成：功能变异模块：从恶意软件源代码文件中提取功能函数，并借助 LLM 进行修改。变种合成模块：将修改后的函数更新至恶意软件源代码，通过编译项目生成变种文件

Algorithm 1，专为功能变异模块设计，详细说明了三个子模块 Extractor、Prompt Generator 和 LLM-Based Function Modifier 如何转换恶意软件源代码中的函数。该算法以文件名 i、要修改的函数数量 j、期望的转换策略 s 以及选定的 LLM 作为输入。接下来，我们将详细描述每个子模块。

算法 1: 使用 LLM 转换函数

输入: 文件名 i，需要改变的函数数量 j，转换策略 s，语言模型 LLM
输出: 转换后的函数集 $\hat{F}_s = \{\hat{f}_1^i, \hat{f}_2^i ..., \hat{f}_j^i\}$

Headers,globals,functions$\{f_1^i, f_2^i, ..., f_G^i\} \leftarrow extractor(i)$;
初始化转换后的函数集 $\hat{F}_s \leftarrow \emptyset$;
for $t \leftarrow 1$ **to** j **do**
    $p_s || f_t^i \leftarrow gen\_prompt(s, f_t^i, headers, globals)$;
    Transform function: $\hat{f}_t^i \leftarrow LLM(p_s || f_t^i)$;
    Update set: $\hat{F}_s \leftarrow \hat{F}_s \cup \{\hat{f}_t^i\}$;
return $\hat{F}_s$

4.1.1 Extractor 子模块

Extractor 子模块（算法 1 的第 1 行）利用 extractor 子程序，该子程序接收一个源文件并遍历源代码的解析树。它从解析树中提取并存储以下两条辅助信息：全局声明的变量、结构体、编译器指令的列表，并将它们存储于 $globals$ 中；以及所包含头文件的列表，并将它们存储于 $headers$ 中。此类信息对于成功转换至关重要，因为它提供了函数可能使用的全局依赖项的基本上下文。以提示的形式将此上下文提供给 LLM 可确保生成更准确且语法正确的代码。此后，该子程序解析源文件以提取所有函数定义，生成集合 $\{f_1^i, f_2^i, ..., f_G^i\}$。

4.1.2 Prompt Generator 子模块

算法 1 的第 3−7 行对应于 Prompt Generator 和 LLM Transformation 子模块。第 4 行中的子程序 gen_prompt 被调用，参数为函数 f_t^i、转换策略 s 以及提取的 $headers$ 和 $globals$。它将输入代码和策略构造成一个为 LLM 定制的提示 $p_s||f_t^i$。提示的设计详见第 IV-C 节。另请参阅附录 F 了解该子程序中使用的不同类型的提示，以及附录 G 了解一个完整构建的提示及其相应 LLM 响应的示例。

4.1.3 LLM Based Function Modifier 子模块

算法 1的第 5 行将设计好的提示 $p_s||f_t^i$ 提供给选定的 LLM，并获取转换后的函数。在代码生成过程中，我们使用了 LLM 的默认推理设置。具体而言，temperature=0.8，top-k=40，top-p=0.9。我们在附录 A-A 中提供了使用 LLM 进行代码生成过程的详细描述。

最后，第 6 行将转换后的函数 f_t^i 追加到输出集合中。一旦所有选定的函数处理完毕，该算法即返回转换后的集合。我们注意到，该算法可以执行多次，以从同一源文件生成函数的多个变体。然而，在本工作中，对于每个选定的恶意软件样本，我们将评估限制在转换后函数的单一版本上。

Algorithm 2，在变种合成模块中实现，使用由算法 1 产生的转换后函数集合 $\hat{F}_s$、恶意软件项目 P 以及被修改的文件 i。它以增量方式生成恶意软件变体，并结合手动调试以确保成功编译。第 1 行初始化恶意软件变体的结果集合 M_s。该集合包含针对文件 i 使用策略 s 生成的恶意软件变体。尽管我们展示了针对某个特定文件的算法，但当我们处理后续文件时，先前处理过的文件的所有修改都会被保留并向前传递，从

而确保恶意软件代码库的累积式转换。该算法的核心功能封装在第 2-10 行中，其中每个转换后的函数被迭代地集成和调试。

算法 2: 恶意软件变种生成

输入: 恶意软件项目 P，文件名 i，转换后的函数集合 $\{\hat{f}_1^i, \hat{f}_2^i ..., \hat{f}_j^i\}$
输出: 编译后的恶意软件变种集合 M_s

```
1 初始化集合: M_s ← ∅;
2 for t ← 1 to j do
3     Extract subset of functions: F̂_t ← {f̂_k^i ∈ F̂_s | 1 ≤ k ≤ t};
4     Generate updated file: î ← merger(i, F̂_t);
5     while compile(P) fails do
6         Debug project P and resolve errors;
7     Compile project: M̂_s ← compile(P);
8     Add compiled malware: M_s ← M_s ∪ M̂_s;
9 return M_s
```

4.1.4 Merger 子模块

第 3 行首先提取函数子集 $\hat{F}_t$，该子集包含函数 1 到 t。下一行使用 $merger$ 子程序，用集合 $\hat{F}_t$ 更新文件 i。它将更新后的函数集成到文件 i 中，同时保持其余函数不变，并利用在 LLM 使用算法 1 进行代码生成过程中的各种簿记信息。合并后，我们获得包含 $(1...t)$ 个修改后函数的更新文件 $\hat{i}$。$merger$ 子程序的更多细节详见附录 A-B。

4.1.5 Compilation and Debugging 子模块

算法 2 的下一步涉及将更新后的文件 $\hat{i}$ 放入恶意软件项目 P 中。第 6–9 行编译更新后的恶意软件项目。如果编译成功，则将生成的恶意软件变体 $\hat{M}_s$ 添加到结果集 M_s 中。如果编译失败，则进行手动调试以解决错误。

手动调试由一位从事对抗性恶意软件领域和恶意软件分类研究的研究人员执行，这确保了修正的一致性和技术可靠性。调试过程严格解决语法错误、构建和项目配置问题，例如链接外部库或更改语言版本，以及恢复 LLM 遗留的不完整的占位符代码。未对 LLM 生成代码的语义逻辑进行任何更改。代码修正在成功编译修改后的恶意软件源代码的前提下，被刻意限制在最小干预范围内。值得注意的是，调试过程专注于第 t 个函数，因为先前 $(1, ..., t-1)$ 的 LLM 生成函数已经过调试和修正，确保错误不会跨迭代传播。调试完成后，编译成功的恶意软件变体可执行文件被添加到 M_s 中。该

过程持续增量进行，直到所有函数处理完毕并返回最终的恶意软件变体集合。

4.2 B. 代码转换策略

我们介绍用于通过 LLM 操作 C/C++ 恶意软件源代码的源代码转换策略。

4.2.1 代码优化

该策略通过提示优化源代码，方法是消除冗余、解决性能瓶颈以及简化代码逻辑，而不改变其核心功能。它涉及使用替代的数据结构和算法，或利用现代库和特定语言特性，例如 C++ 算法头文件中的搜索函数。这些优化可能改变代码的执行和性能特征，有可能降低静态或基于启发式方法的检测率。

4.2.2 代码质量与可靠性

该策略确保生成的代码遵循标准实践，具有改进的错误处理并解决边缘情况。额外的错误处理可防止执行期间的运行时问题，并为代码添加分支，这使恶意软件更加可靠。

4.2.3 代码可重用性

该策略侧重于将大型函数拆分为更小的模块化块。这些较小的函数调用通过改变执行流，有助于掩盖恶意软件的真实行为，这使得依赖涉及控制流执行模式的检测器在实现恶意软件预期相同结果的同时面临更大挑战。

4.2.4 代码安全性

该策略通过遵循安全编码标准来解决潜在的安全漏洞。恶意软件（如勒索软件）严重依赖加密库进行数据加密和解密。此方法提示 LLM 用替代方案替换这些库，修改敏感操作的实现，同时保持恶意软件的核心功能。通过混淆加密行为，检测引擎可能更难以识别该可执行文件为恶意软件。

4.2.5 代码混淆

该策略通过使代码更难以分析和逆向工程来增强恶意软件的规避能力。它包括用无意义的名称重命名函数和变量、添加不必要的控制流结构（例如，跳转、循环）、改变现有控制流以及插入反调试技术。它还定义并调用多余函数，并添加仅在罕见条件下触发的执行路径。这些转换旨在使静态和动态分析复杂化，同时保留恶意软件的核

心功能。

4.2.6 Windows API 特定转换

该策略使用提示来识别恶意软件函数内的 Windows API 调用，并引导 LLM 用替代或间接等效的 API 替换它们。它还可能引入包装函数以模糊直接的 API 使用。我们不是使用静态映射，而是利用 LLM 的生成能力来产生多样化的 API 替换，从而增加变异性并避免预定义映射的僵化性和可扩展性问题。尽管功能保持不变，但改变后的 API 模式可能会混淆基于启发式的检测系统（这些系统通常依赖常见的 Windows API 使用模式），从而使恶意软件更难以检测。

4.3 C. LLM 的快速设计

在本节中，我们描述用于生成提示的 Algorithm 3。我们还介绍了 LLM 在转换给定函数 f_t^i 时必须遵循的约束。

它基于给定的转换策略 s、第 t 个函数 f_t^i 以及文件 i 的 $headers$ 和 $globals$（如算法1中所定义）进行操作。该子程序以调用 $system\_prompt$ 开始，生成 p_{sys}。这定义了 LLM 作为专业编码助手的角色，该助手在系统编程以及 C、C++ 和 C# 等语言方面拥有专业知识，并确保代码转换是在适当的上下文和能力范围内进行的。随后，$intro\_prompt$ 通过接收目标函数 f_t^i 的名称生成 p_{intro}，并指定必须使用以下预定义策略将提供的函数以及必要的头文件和全局变量转换为变体函数。接下来，使用 $strategy\_prompt$ 和 s 生成转换策略提示 p_{strat}。这些步骤建立了转换上下文并指导 LLM 执行所需的修改。

算法 3: 基于 LLM 实现函数转换的快速构造子程序

Function $GEN\_PROMPT(s,\ f_t^i,\ headers,\ globals)$**:**
- $p_{sys} \leftarrow system\_prompt()$;
- $p_{intro} \leftarrow intro\_prompt(f_t^i.name)$;
- $p_{strat} \leftarrow strategy\_prompt(s)$;
- $p_{pres} \leftarrow preserve\_rules\_prompt(f_t^i.name)$;
- $p_{addit} \leftarrow additional\_prompt(f_t^i.name)$;
- $p_{code} \leftarrow headers \oplus globals \oplus f_t^i$;
- $p_{user} \leftarrow p_{intro} \oplus p_{strat} \oplus p_{preserve} \oplus p_{additional} \oplus p_{code}$;
- return $p_s || f_t^i = p_{sys} \oplus p_{user}$;

保留提示 p_{pres} 试图确保原始函数和转换后的函数在语义上等价。它明确指示模

型不要修改全局定义或自定义元素（变量、对象、常量）以保持功能一致性，从而避免整个代码库中可能出现的语法或语义错误。此外，p_{addit} 施加严格的准则以保持一致的、符合语法的代码格式并保留函数签名，指示模型仅在单个特定语言的代码块中生成修改后的函数和必要的头文件，以便于解析和后处理。这确保了输出是完整的，不遗留未完成的代码块，并避免重新生成原始函数。

在完成这些步骤后，代码的提示通过组合 $headers$、$globals$ 和函数 f_t^i 的定义来构建，其中符号 $\oplus$ 代表字符串连接。这使我们能够通过连接第 3–7 行的所有提示来构建总的用户提示 p_{user}。然后，通过连接 p_{sys} 和 p_{user} 来构建最终提示 $p_s||f_t^i$。这种结构化的方法确保 LLM 接收到用于转换任务的明确且完整的指令，同时满足所有必要的约束和要求。

表 4.1　选择的恶意软件样本总结

样本	语言	LOC	文件数量	函数数量	VT 率	HA 率	类型
Exeinfector	C++	230	1	4	72.009	26.67	感染型病毒
Fungus	C++	2266	15	46	73.630	76	通用犯罪软件
Dexter	C	2661	12	61	83.020	88	POS 木马
HiddenVNC	C++	4959	18	60	76.503	75	HVNC 机器人
Predator	C++	4145	10	102	58.797	70.333	信息窃取
Prosto-Stealer	C++	7436	27	143	62.033	72.333	信息窃取
Conti(Cryptor)	C++	8031	35	99	65.275	79.333	勒索软件
Babuk(Cryptor)	C++	3910	22	62	71.759	83.667	勒索软件
RedPetya	C++	1494	5	15	62.500	56.333	勒索软件
RansomWar	C	1357	5	13	65.728	50.333	勒索软件

第 5 章 评估

在本节中，我们进行全面评估以回答以下问题：• RQ1：LLMalMorph 生成的恶意软件变体对广泛使用的防病毒引擎和机器学习分类器的检测具有多强的抗性？其规避能力与最近的对抗性恶意软件生成框架生成的变体相比如何？• RQ2：不同转换策略间的代码编辑工作量有何差异？这对揭示 LLM 所犯错误类型有何启示？• RQ3：生成的恶意软件变体是否保留了原始样本的语义和功能？

5.1 评估设置

5.1.1 选择样本

由于最新的恶意软件源代码稀缺，大多数恶意软件研究都集中在可执行文件上。我们检查了公共数据库 [38], [39]，发现大多数可用的 Windows 恶意软件源代码是 32 位的，因此我们将研究重点放在 32 位变体上。我们选择了能够编译成可运行可执行文件、并表现出可通过 VirusTotal 或 Hybrid Analysis 检测到的恶意行为（其 AV 检测率 ≥60%）的样本。这产生了十个具有不同复杂性和类型的恶意软件候选样本。RansomWar 样本使用 GCC 编译，其余样本使用 Microsoft Visual Studio 2022[1]编译（因为提供了.sln 文件）。表 II 总结了选定样本的关键细节。对于 Conti 和 Babuk 勒索软件，我们的分析集中在负责加密的加密器组件上。有关样本的详细信息请参见附录 B。

5.1.2 评估指标

我们使用给定的评估指标：防病毒（AV）检测率（$R^{\hat{M}s}$）。我们使用 VirusTotal 评估 AV 检测率，该平台使用来自不同供应商的多个 AV 引擎扫描样本。每个样本可用的检测器数量因可用性而异。令 D 为可用检测器的集合，$\hat{D} \subseteq D$ 为将恶意软件变体标记为恶意的检测器集合。在第 k 次运行时，变体 $\hat{M}_s$ 的 AV 检测率 $R_k^{\hat{M}_s}$ 定义为 $R_k^{\hat{M}_s} = \frac{|\hat{D}|}{|D|} \times 100$，其中 $|.|$ 表示两个集合的大小。

每个样本将进行 k 次运行，并计算平均检测率 $R^{\hat{M}_s} = \frac{1}{k}\sum_{i=1}^{k} R_k^{\hat{M}_s}$。我们在实验中设定 k=3 以解释不同运行间的变异性。我们还使用 Hybrid Analysis，该工具包含静态分析、基于机器学习的分析以及使用不同引擎的多重扫描分析。检测性能以每个恶意软件样本及其变体在 k 次运行上的平均百分比表示。我们使用这两个工具的免费版

[1]https://visualstudio.microsoft.com/vs/

本，并通过其各自的 API 自动化样本上传、结果检索和后处理。

5.1.3 策略导向的机器学习分类器攻击成功率 (ASR)

攻击成功率 (Attack Success Rate, ASR) 是评估对抗攻击的广泛使用的指标 [20], [40]，即生成的恶意软件变体成功逃避目标系统检测的比例。我们针对三种基于机器学习的恶意软件分类器评估 ASR：Malconv [41]、Malgraph [42] 和一个训练好的 ResNet50 恶意软件分类器 [43]。模型细节详见附录 E。令 M 为一个原始恶意软件样本，将我们的策略 s 应用于所有修改文件 $\hat{F} \subseteq F$ 并为 j 个转换函数生成变体 $V_s^M = \{\hat{M_1}, \hat{M_2}, ..., \hat{M_j}\}$。对于给定的目标分类器 C，令 $V_{s,C}^{\hat{M}} = \{\hat{M} \in V_s^M : C(\hat{M}) = benign\}$ 为成功逃避 C 的变体子集。则攻击成功率为 $ASR = \frac{|V_{s,C}^{\hat{M}}|}{|V_s^M|} \times 100$。其中 $|.|$ 表示集合的大小。

5.1.4 策略导向的代码编辑工作量 (W_s^M)

该指标基于需要手动编辑以编译 LLM 生成代码的总代码行数来比较策略 s。值越高表明需要更广泛的手动干预，意味着 LLM 在该策略下犯了更严重的错误。对于给定的恶意软件 M，令 $\hat{F} \subseteq F$ 表示 F 个文件中被修改的文件数量。恶意软件 M 和策略 s 的编辑工作量 W_s^M 定义为在文件 $\hat{F}$ 中所有转换后的函数 $\{f_1^i, f_2^i ... f_j^i\}$ 上编辑（添加、修改或删除）的总代码行数，计算公式为 $W_s^M = \sum_{i=1}^{\hat{F}} \sum_{t=1}^{j} L_{edit}(f_t^i)$，其中 L_{edit} 统计函数 $\hat{f_t^i}$ 被编辑代码的行数。

5.1.5 人力投入量化指标 (H_s^M)

该指标衡量针对每种策略调试和配置恶意软件中转换后函数所需的人力投入。它量化了生成一个成功编译的 PE 文件所需的总工时。对于给定的恶意软件 M 和代码转换策略 s，人力投入 H_s^M 的计算公式为 $H_s^M = \sum_{i=1}^{\hat{F}} \sum_{t=1}^{j} ManHours(\hat{f_t^i})$。此处，求和表示在策略 s 下，在修改文件 $\hat{F}$ 中所有转换后的函数 $\{f_1^i, f_2^i ... f_j^i\}$ 上所花费的总工时。

5.1.6 功能保留指标 (Φ^M)

该指标评估了经 LLMalMorph 转换后，恶意软件语义在变体中保留的程度。由于可执行文件固有的复杂性，尚无精确方法能判断恶意软件 M 与其变种 $\hat{M}$ 之间的语义等价性 [44]。因此，文献中遵循的评估方法包括比较恶意软件与其变体之间的 API 调

用序列 [22]，或通过在沙箱中运行来比较恶意软件及其变体的行为 [20], [23]。

我们采用类似方法，并利用最长公共子序列 (lcs) 算法来比较 M 和 $\hat{M}$ 之间的 API 调用序列。转换后的变体必须保留原始的 API 调用顺序，允许存在不破坏此序列的额外调用。API 调用序列使用专有沙箱收集。归一化的最长公共子序列定义为 $\hat{lcs}(M,\hat{M}) = \frac{lcs(M,\hat{M})}{Length(API(M))}$，其中分母是 M 的 API 序列长度。分数范围从 0 到 1，1 表示完全相同的 API 序列。最后，我们计算功能保留率 $\Phi^M = \frac{|\hat{M}\in\psi^{\hat{M}}:\hat{lcs}(M,\hat{M})\geq\delta|}{|\psi^{\hat{M}}|} \times 100$。

此处，$\psi^{\hat{M}}$（总变体集）定义为所有 AV 检测率 $R^{\hat{M}_s}$ 低于 M 的基线检测率的恶意软件变体 $\hat{M}$ 的集合。分子代表 $\psi^{\hat{M}}$ 中保留了语义等价性的子集的大小，我们通过考虑归一化的最长公共子序列得分（其值大于预定义阈值 δ）来确定该子集，而 $|\psi^{\hat{M}}|$ 是整个总变体集的大小。我们通过经验分析恶意软件变体和原始样本，选择 δ 的值为 0.96。我们选取不同样本在离散值集上的恶意软件变体，并将其上传到 Triage Sandbox[2]。我们分析了变体和原始恶意软件样本的报告，以比较行为指标、注册表修改、网络调用等。我们观察到，在某些情况下，行为漂移在分数低于 0.96 时开始出现，而在其他情况下，即使分数略高于该值，功能等价性也得以保持。然而，大多数保留了关键行为的变体得分在 0.96 或以下。因此，我们选择 $\delta = 0.96$ 作为上限，以确保被接受的变体在 API 序列和执行行为上保持高度相似性。

5.2 模型选择

虽然 LLMalMorph 可以利用任何 LLM 生成恶意软件变体，但我们选择 Codestral-22B [8] 作为我们的主要 LLM。我们精心设计的提示包含许多需要遵循的约束和指示。我们观察到 Codestral 模型能够比其他模型更精确地遵循这些特定指令来变异函数。此外，Codestral 为我们的用例提供了一组均衡的特性——220 亿参数、12 GB 模型大小和 32K 上下文窗口——并且与具有更高硬件要求的模型相比，它在长距离仓库级代码补全任务上具有卓越性能 [4], [9]。

5.3 实现细节

在 LLMalMorph 中，Function Mutator 内的 Extractor 子模块使用 Tree-sitter Parser[3] 的 Python 绑定实现，利用其基于 C 的运行时进行高效解析。对于 LLM，我们使用

[2]https://tria.ge/

[3]https://tree-sitter.github.io/tree-sitter/

了 Ollama[4]，它便于本地 LLM 执行而无需外部 API 依赖，并提供了一个基于 Python 的接口。我们的实验设置包括一台配备 252 GB RAM 和 48 个处理器的 RTX 3090 GPU 服务器，以及一台配置了 VirtualBox[5] 用于恶意软件编译的 Windows 10 虚拟机。LLMalMorph 的核心实现主要使用 Python 开发，部分组件（例如基于 lcs 的语义指标计算）使用 C++ 实现。

5.4 评估结果与分析

我们使用算法 1 和 2 对要修改的文件进行优先级排序，按函数数量升序排列。这假设对源代码了解有限的攻击者会针对函数较少的文件，以最小化工作量并最大化修改文件的数量。如果出现平局，则随机选择文件。我们依次修改每个文件内的函数，但这可能会忽略对恶意软件行为至关重要的关键函数。自动化分离恶意函数具有挑战性，因为看似良性的函数（例如，简单的线程管理）可能促成恶意活动。我们的目标是评估这种简单的顺序方法是否能在不损害功能的情况下产生规避性变体。有关文件选择标准以及每个样本选择要修改的函数数量的标准，请参见附录 C。

5.4.1 对研究问题 1（RQ1）的解答

我们评估了恶意软件变种逃避杀毒软件检测器的有效性。图 2a 与图 2b 展示了 10 个恶意软件样本在 VirusTotal 和 Hybrid Analysis 平台的检测率。六种代码转换策略以不同颜色标注，标记点表示被修改的文件。每个数据点代表特定策略的检测率；x 轴按递增顺序显示被修改的函数数量（例如 3 代表函数 1–3），y 轴则绘制杀毒软件检测率（$\hat{R}_k^{M_s}$）。黑色虚线标示各样本的基准检测率，红色虚线显示所有变种的平均检测率。此外，我们展示了针对 Malgraph 分类器的四个样本通过不同策略实现的攻击成功率（ASR），并简要讨论了与最新对抗性恶意软件生成框架的对比分析。

5.4.2 VirusTotal

图 2a 显示，10 个样本中有 5 个样本的所有恶意软件变种检测率均低于各自基准值。对于 Exeinfector，其变种平均检测率为 40.708%，比基准值 72.009% 低 31.301%，且检测率呈下降趋势。最显著的下降发生在修改第 4 个函数后，此时 Reusability、Optimization 和 Security 策略的检测率均跌破 30%，较基准值下降超 42%。在 Fungus

[4]https://ollama.com/

[5]https://www.virtualbox.org/

样本（第 2 个子图）中，平均检测率为 63.167%，而基准值为 73.630%。Optimization 策略在修改三个函数（包括文件 mUsb 中操纵 USB 驱动器创建隐藏目录并自动执行文件的第三个函数）后达到最低检测率 56.611%。在图 2a 的 Dexter 子图中，检测率平均值为 72.211%，比基准值 83.020% 低 10.809%。该样本的详细数据见附录 D。对于 HiddenVNC 样本，平均检测率为 64.664%，较基准值 76.503% 低 11.84%。Optimization 与 Quality 变种检测率持续较低，而 Security 与 Windows 变种呈下降趋势；Reusability 策略在第 7 次至第 8 次函数修改期间从 67.593% 显著降至 61.081%。被修改的第八个函数被拆分为六个更小的函数，用于枚举可见窗口并捕获其内容。

对于 Predator 样本，平均检测率与基准值几乎相同，多数变种表现相似。但 Security 策略呈下降趋势，其在第 8 个函数处达到最低检测率 49.967%（较基准值低 8%），Optimization 策略紧随其后为 54.591%。Prosto 样本的详细数据见附录 D。在 Conti 勒索软件样本中，大多数变种检测率接近 65.275% 的基准值，但 Reusability 策略在第 7 个函数处出现大幅下降，Optimization 策略在第 5 个函数处同样大幅下跌——二者起始高于基准值但之后持续低于该值。Windows 策略与 Quality 策略亦呈现下降趋势。对于 Babuk 样本，多数策略的检测率出现中等程度下降，其中 Optimization 策略在早期阶段造成最显著的跌幅。详细分析见附录 D。

对于 RedPetya 样本，我们观察到 Security 与 Quality 变种的检测率呈下降趋势。值得注意的是，Security 策略在第 8 个函数处出现最急剧下降至 46.746%，较基准值降低 15.75%。该下降源于 LLM 使用了替代 OpenSSL 的加密库。被修改的第 8 个函数 $hard\_reboot$ 通过调用 Windows API 调整进程权限并触发重启，使勒索软件能在下次启动时重获控制权，这是关键持久化机制。在 RansomWar 样本中，50.251% 的平均检测率较 65.728% 的基准值低 15.478%，所有变种均跌破基准值。多数变种检测率位于 50-58% 区间，而 Optimization 策略在第 3 个函数修改后跌至 34.5%。Windows 策略以 28.651% 的检测率（低于基准值 37%）达到最低值，展现出强规避能力。

整体而言，Optimization 策略在降低恶意软件检测率方面表现最为稳定，其次为 Security 与 Reusability 策略，而 Windows 策略虽在各样本中展现出强规避能力但效果波动较大。Optimization 策略通过引入和重组数据结构，结合可能改变控制流与数据流的特定语言特性，从而修改了杀毒引擎通常可检测的代码模式与语义。这些特定修改需要新增头文件及库引用，进而引发编译后二进制文件的变化，最终改变其在杀毒检测器中的识别蓝图。Security 策略采用替代性加密库可导致更显著的二进制差异，

破坏杀毒软件基于模式的启发式签名机制，从而产生低于基准值的检测率。

5.4.3 混合检测

在图 2b 的 Exeinfector 子图中，Optimization 与 Security 策略在第二次函数修改后呈现下降趋势。混淆策略在修改第二个函数后将检测率提升至 41%，这可能源于引入了已知的反调试函数。类似地，Windows 策略在两次及四次修改后显示更高检测率，表明 LLM 生成的替代性 API 调用更易被检测到。在 HiddenVNC 的第四个子图中，Quality 策略对第 10 个函数添加错误检查后达到最低检测率 67.333%。Security 策略中，添加到第 10 个函数的基于 OpenSSL 的功能会暂时降低检测率，之后出现回升；Optimization 子图也观察到类似情况。在第二个 Fungus 子图中，所有变种的平均检测率为 66.636%，较 76% 的基准值低 9.364%，其中 Optimization 策略实现最低检测率并生成最具规避性的变种。对于 Dexter 样本，88% 的基准检测率降至 80.653% 的平均值。Windows 策略在四次函数修改后达到最低检测率，这些修改涉及 LLM 生成的代码转换，包括替换基于注册表修改的函数，以及使用 $ZeroMemory()$ 和 $lstrlenA()$ 实现安全内存处理与字符串长度计算。

对于第 5 至第 9 个子图，检测率出现骤升现象，这是因为 Hybrid Analysis 有时跳过杀毒软件检测环节，仅依赖机器学习与静态分析，导致分数虚高（例如 100/90）并扭曲平均值。Predator 与 Prosto 样本整体变化极小，而 Conti 样本出现显著的 Optimization 策略特异性下降；这三个子图（5-7）的详细分析见附录 D。在 Babuk 样本中，Optimization 策略持续降低检测率，最低达 74%，较基准值 83.667% 下降约 10%，该趋势与图 2a 一致。Quality 与 Obfuscation 策略同样呈现下降趋势。对于 RedPetya 样本，Security 策略在第 9 个函数处降至 45.333%，在第 8 个函数处为 46.0%，与图 2a 趋势一致，证明 LLM 在函数转换中的有效性。在 RansomWar 样本中，变种平均检测率 36.697% 较基准值 50.333% 低 13.636%。Security 策略保持稳定的 10% 检测率，而 Windows 策略以 6% 的检测率达到最低值，较基准下降 44%。

这些发现印证了 VirusTotal 的结果：Optimization、Security 与 Windows 策略对降低检测率贡献最大。如前所述，Optimization 策略可能重塑控制流与数据流，而 Security 策略采用替代性加密库会向修改后的恶意软件二进制文件引入新导入项和符号。这不仅影响基于签名的检测器，也可能影响混合检测使用的静态与机器学习检测器。值得注意的是，对于两个检测平台，检测率下降程度与被修改函数的数量无关，且可能产

生相反效果。

5.4.4 机器学习分类器

表 5.1　各策略的 ASR(%)。百分比已对不同变体数量进行标准化（Fungus：9 个变体/策略；Dexter：12 个；Conti：14 个；Babuk：11 个）。

样本	Optimization	Quality	Reusability	Security	Obfuscation	Windows
Fungus	88.889	0	11.111	0	0	0
Dexter	50.00	16.667	0	41.667	33.333	0
Conti	71.429	0	0	0	0	0
Babuk	0	72.727	0	90.909	0	0

针对每个 ML 模型，我们选择其对应的 0.1% 误报率（FPR）阈值作为检测阈值。Malconv 与 ResNet50 模型未将原始 10 个样本中的任何样本识别为恶意软件，而 Malgraph 仅标记了 Fungus、Dexter、Conti 和 Babuk 样本。因此我们仅展示这四个样本在 Malgraph 上的检测结果。阈值设定细节详见附录 E，攻击成功率（ASR）数据列于表5.1：

第一列列出四个样本名称，后续列展示六种转换策略的攻击成功率（ASR）。Optimization 策略在第一和第三样本中实现高 ASR，在第二样本中取得中等成功率；Security 策略在 Babuk 样本呈现高成功率，在 Dexter 样本为中等成功率，这与杀毒检测器的观测结果一致。Babuk 样本在 Quality 策略下同样表现出显著成功率，与其在 Hybrid Analysis（图 2b）中的行为一致。Reusability 与 Obfuscation 策略在两个样本中成功率较低，而 Windows 策略未能规避检测。

尽管 LLMalMorph 通过提示词工程和精选转换策略修改函数（未直接针对任何 ML 恶意软件分类器进行优化），仍在部分样本的 Optimization 与 Security 等策略上展现出显著 ASR，这支持了杀毒检测率降低的观测结论。正如先前在杀毒检测器语境中指出的，这些策略的高 ASR 可能源于 LLM 使用新库、新特性及控制流重组，这些改变很可能促进了规避成功。

5.4.5 比较分析

现有大多数对抗性恶意软件生成研究（包括 Malguise[22] 等最新框架）集中于直接修改已编译的二进制文件以生成规避性变种。因此，我们将 LLMalMorph 与领先的二进制级对抗性恶意软件生成框架 Malguise 进行对比。如研究问题 1（RQ1）解答

中机器学习分类器小节所述，仅有 4/10 样本（Fungus、Dexter、Conti 和 Babuk）被 Malgraph 分类器识别为恶意软件，其他分类器没有任何样本被判定为恶意。我们在上述四个样本上运行 Malguise[22]，以 Malgraph 为规避目标，为每个样本生成对抗性变种。Fungus、Dexter 和 Babuk 成功绕过 Malgraph 分类器，而 Conti 未能实现规避。尽管如此，我们收集了 Malguise 生成的所有对抗性变种（三个成功规避样本和一个未成功样本），通过 VirusTotal 与 Hybrid Analysis 运行并记录杀毒检测率指标（该指标定义见第五节 A.2 评估部分的评估设置小节）。结果呈现在表5.2中。

表 5.2　VirusTotal 和混合分析对于 Malguise 和 LLMalMorph 生成的对抗变种 AV 检测率（%）的比较。

恶意软件	VirusTotal-Malguise	VirusTotal-LLMalMorph	混合分析-Malguise	混合分析-LLMalMorph
Fungus	61.574	63.167	69.667	66.636
Dexter	78.241	72.211	83.333	80.653
Conti	66.667	63.667	75.667	71.568
Babuk	72.685	70.326	82	80.025

表5.2的第一列为本次对比分析使用的四个恶意软件样本。随后两列展示 VirusTotal 平台检测率数据（分别对应 Malguise 与 LLMalMorph 变种），最后两列呈现 Hybrid Analysis 的对比数据。对于 LLMalMorph，我们展示每个样本所有生成变种的平均杀毒检测率（即图 2 中红色虚线所示）。在 VirusTotal 数据中，Fungus 样本的检测率与 Malguise 接近，其余样本杀毒检测率更低。观察到除 Fungus 外的三个样本检测率降幅最高约 6.03%，平均降幅约 3.8%。在 Hybrid Analysis 数据列中，所有样本均呈现比 Malguise 更低的下降幅度：该杀毒引擎检测率最大降幅约 4.1%，平均降幅约 3%。

该分析为基于源代码的 LLM 驱动转换能力提供了深刻见解。Malguise 在二进制层面运作，通过语义 NOP 插入和基于调用的重划分技术修改已编译可执行文件，生成专为规避机器学习分类器优化的恶意软件变种。该方法无需重新编译，可直接修补二进制文件而非修改源代码。相比之下，我们的框架利用 LLM 在源代码层面转换恶意软件，需经编译与调试以保持功能正确性。我们未采用基于搜索的方法，也未针对任何目标（如 Malguise 中的分类器）进行优化。尽管存在根本性方法论与抽象层级差异，LLMalMorph 生成的变种仍取得了与 Malguise 具竞争力的杀毒检测率。虽然检测率下降幅度微小，但 LLMalMorph 在未显式优化规避能力的情况下表现相当，这证明了 LLM 引导的恶意软件源代码转换在生成规避性变种方面的实用潜力。

5.4.6 研究问题 2（RQ2）的解答

所有样本各策略的代码编辑工作量通过图 3 的雷达图展示，每个顶点代表特定恶意软件 M 在给定策略 s 下的代码编辑工作量 (W_s^M)。类似地，图 4 以工时（H_s^M）呈现人力投入。图表中策略名称采用缩写形式以提高可读性，完整名称详见图例。工作量统计包含 LLM 建议回填至函数的代码修改量——由于要求生成完整代码，当 LLM 建议在函数中回填特定代码行时，这些回填内容计入修改量。

5.4.7 代码编辑工作量

Exeinfector 雷达图（图 3）显示：Quality 策略工作量最高达 23 行，Obfuscation 次之为 9 行，Security 为 7 行，而 Windows 与 Reusability 均低于 2 行。Fungus 图中 Obfuscation 工作量最高（117 行），其 Windows 与 Reusability 工作量较 Exeinfector 增加，但 Optimization 与 Security 降低。Fungus 因函数复杂度高导致 LLM 错误增多，需手动回填代码。Dexter 与 HiddenVNC 样本中，Optimization 策略工作量最高（Dexter 达 154 行，HiddenVNC 为 85 行），尽管二者在图 2a 中检测率呈下降趋势。Dexter 的 Reusability 与策略另占 88 行，其高 Optimization 错误源于 LLM 生成代码引发编译问题后需重用原始函数。HiddenVNC 的 Optimization 策略因需添加大段代码导致工作量最大（Quality 策略次之 54 行），其余策略均 ≤22 行。Predator 样本中 Windows（31 行）与 Security（37 行）编辑量最高，其他策略 <20 行；Security 因 LLM 错误使用”BCryptDecrypt” 需重用原始代码。Prosto 样本 Optimization（27 行）与 Security（26 行）工作量最高，其余策略 <12 行。Conti 勒索软件中 Obfuscation（18 行）与 Windows（20 行）代码行数最多，其他 <10 行；Windows 策略虽明确要求 LLM 生成完整代码，仍需回填 20 行。Babuk 样本 Security（8 行）、Obfuscation（10 行）、Windows（8 行）工作量相对较高，其余策略修改量极小。RedPetya 与 Prosto 模式相似，Security/Windows/Optimization 占主导但均 <20 行。RansomWar 样本 Windows 工作量最高（38 行），需构建封装器支持 LLM 生成函数，并为 LLM 遗漏函数回填动态 DLL 加载代码。

整体而言，Code Optimization、Windows 与 Security 策略在多数样本中导致高工作量，其余策略工作量则波动不定，反映出 LLM 处理不同策略及样本复杂度的差异性。高工作量主要源于重用原始函数、生成不完整代码、以及错误使用复杂 Win32 API 与替代性加密库调用。尽管成本高昂，Optimization 与 Security 策略在杀毒和机器学习检测中展现出最强规避能力，突显了成功概率与人力投入之间的权衡关系。

5.4.8 人力投入

图 4 显示 Exeinfector 与 Fungus 样本模式相似：Windows 策略耗时最高（Exeinfector 为 0.3 工时，Fungus 达 1.333 工时）。Exeinfector 样本中 Security 与 Quality 策略分别需 0.15 和 0.2 工时，Optimization 耗时 0.13 工时，其余策略耗时极低。Fungus 样本 Windows 策略高耗时源于调试”mUsb” 函数；其他显著耗时包括 Obfuscation（0.517 工时）、Reusability（0.433 工时）及 Optimization（0.317 工时）。Optimization 作为降低检测率最有效的策略之一（见图 2a、2b），其生成可编译执行文件所需工时较少，故在该二样本中具备高效生成恶意软件变种的优势。Dexter 样本各策略工时分布近似均匀：Optimization（0.55 工时）、Windows（0.633 工时），其余如 Obfuscation/Security/Reusability 介于 0.4-0.42 工时。HiddenVNC 样本则波动较大：Security（0.717 工时）、Optimization（0.667 工时）、Windows（0.45 工时）。其 Security 策略高耗时源于 LLM 集成 OpenSSL 至现有代码库的困难，反映出 LLM 修改安全相关库的内在挑战。Predator 子图中 Reusability（0.267 工时）、Security（0.383 工时）、Obfuscation（0.2 工时）耗时最高；Security 策略因调试”BCryptDecrypt” 函数需额外耗时。Prosto 样本 Security 策略最耗时为 0.417 工时，Optimization/Windows/Reusability 分别为 0.267/0.183/0.167 工时；此工作量源于添加 OpenSSL 及 LLM 对其函数处理不当。Conti 勒索软件中除 Reusability（0.23 工时）外，多数策略调试耗时极低。Babuk 样本 Optimization/Reusability/Obfuscation/Windows 策略均约 0.083 工时，分布近似均匀。RedPetya 样本除 Security 策略（0.367 工时，需集成 LLM 使用的”CryptoPP” 库）外整体耗时较低；值得注意的是，该策略在图 2a 与 2b 中亦展现强规避性。末样本模式类同 Babuk 与 Dexter：Windows 与 Security 策略均为 0.233 工时，Reusability 与 Obfuscation 为 0.1833 工时。

整体而言，Windows 策略因 LLM 处理冗长 Windows API 调用存在困难，在所有恶意软件样本中均需高调试工作量。同时，Security 策略因引入新库与新特性、以及语法改动增加，常导致 LLM 在代码生成过程中出错，故需大量投入。

5.4.9 对研究问题三（RQ3）的解答

表 5.3展示了通过公式 1 计算的所有十个恶意软件样本在 VirusTotal 和 Hybrid Analysis 平台上的功能保留指标（Φ^M）。根据图 2a，前四个及最后一个样本变体在 VirusTotal 上具有规避性，因其检测率低于各自基线阈值。在 Hybrid Analysis 中，Fungus、Dexter 和 RansomWar 变体持续保持规避性。Exeinfector 的 Φ^M 在 VirusTotal

表 5.3　所有样本在 2 个反病毒检测器下的功能保留率 Φ^M

样本	VirusTotal	混合分析
Exeinfector	75	72.222
Fungus	31.481	31.481
Dexter	66.667	66.667
HiddenVNC	75	68.182
Predator	36.667	50.0
Prosto	41.667	50.0
Conti	19.565	44.304
Babuk	30.0	37.255
RedPetya	85.714	88.889
RansomWar	55.556	54.902

上高达 75%，在 Hybrid Analysis 上为 72.222%。Dexter 和 HiddenVNC 均表现良好：Dexter 在两个检测器中保持 66.667% 的保留率，HiddenVNC 在 VirusTotal 上为 75%，在 Hybrid Analysis 上约为 68%。Fungus 的 ΦM 较低（约 31.5%），两个检测器中仅 54 个变体中的 17 个维持在检测阈值之下，在两大 AV 检测器上保留了原始语义。Predator 和 Prosto Stealer 呈现中等保留率——VirusTotal 上分别为 36.667% 和 41.667%，二者在 Hybrid Analysis 上均达到 50%。在 Predator 中，LLM 修改了”Stealing.cpp” 中的六个函数；而在 Prosto 中，LLM 对影响目录搜索、文件处理和 Telegram 相关操作的函数进行了跨多文件的广泛编辑。尽管通过调试确保可编译，但 LLM 生成的代码缺乏功能保留性。Conti 的 ΦM 值最低，仅约 20% 的规避变体为 VirusTotal 保留了语义。LLM 修改了关键函数，包括禁用安全钩子、进程白名单和枚举逻辑驱动器的函数。这些修改降低了检测率，但导致功能保留性下降。在 Babuk 勒索软件案例中，其 VirusTotal 保留率与 Fungus 样本相近，而 Hybrid Analysis 保留率更高（37.255%）。RedPetya 以最高保留率（85.714% 和 88.889%）表现突出，证明即使进行如图 2a、2b 所示的安全相关复杂转换，LLMalMorph 仍能成功实现规避的同时保持功能。最终样本的保留率约为 55%。

这些结果表明，虽然优化（Optimization）能有效降低检测率，但在 Fungus、Conti 和 Babuk 等样本中难以保持语义保留。相反，四个样本展现出高 Φ^M 值：Exeinfector、

Dexter 和 HiddenVNC 达到或超过 66%，RedPetya 超过 85%，证明 LLMalMorph 能够生成功能完备且具有规避性的变体。值得注意的是，我们的转换仅在源代码级别操作，但在 Dexter、HiddenVNC 和 RedPetya 等复杂样本中仍能实现高功能保留，验证了 LLMalMorph 中系统化转换和提示设计的有效性。

第 6 章 相关工作

恶意软件变体生成的研究已探索多种方法。大量工作聚焦于全局或局部修改恶意软件的二进制代码，通过在特定位置注入或追加字节而不改变其行为来保持原始功能[18], [45]–[49]。另一种方法涉及二进制多样化技术以全局改变恶意软件的二进制文件[20], [50]。此外，利用贪婪算法、基于梯度的优化、生成模型和启发式技术添加无关函数来操控 API 调用，已成为重要研究领域 [23], [51]–[53]。另有方法通过修改底层汇编代码或使用搜索算法/基于学习的优化在特征空间内改变控制流图 [22], [29]。直接扰动恶意软件代码空间的研究虽较少探索，但包括注入汇编代码以调用外部 DLL 来触发额外 API 而不改变控制流 [54]。Murali 等人 [21] 提出的方法操作 LLVM 生成的中间表示，通过策略性转换直接修改系统调用有向图，随后重新生成恶意软件可执行文件。Choi 等人 [19] 采用名为 AMVG 的自适应框架，通过解析源代码并运用遗传算法生成恶意软件变体。他们在部分 Python 样本和良性 C 程序上展示了简单转换的结果，但仅限于复杂度较低的案例。

第 7 章 结论和未来工作

本研究中，我们提出了 LLMalMorph 框架——该框架利用 LLM 通过工程化提示和代码转换策略生成恶意软件变体。采用 6 种策略生成 618 个变体，证明了特定转换能降低 AV 检测率，并在机器学习分类器上具有显著攻击成功率。同时观察到复杂恶意软件常需大量调试以维持功能，这凸显了人工监督的必要性、提示设计的谨慎性，以及当前 LLM 在恶意软件源代码转换中的局限性。

尽管本研究展示了 LLM 生成规避性恶意软件变体的潜力，仍存在若干局限。我们的实现可通过增强提取器与合并器子模块扩展到其他语言。未来计划拓展至 LLM 二进制恶意软件转换，并利用 LLM 代理提升自动化水平。此外，旨在通过分离恶意软件相关模式改进函数选择机制，并在 API 序列之外开发更鲁棒的语义保留评估指标。

第 8 章 致谢

本研究由思科研究院（Cisco Research）提供支持。我们衷心感谢思科在本研究全程提供的资金支持与宝贵指导。本文表述的任何结论、观点或建议均代表作者立场，并不必然代表资助方的立场。

第 9 章 附录 A

表 9.1 修改的函数选择标准

函数数量	修改率
<10	100%
10 - 20	60%
20 - 40	30%
40 - 70	20%
>70	15%

表 9.2 每个恶意软件样本的函数选择

恶意软件样本	总共选择的函数	修改的函数	修改率
Exeinfector	4	4	100%
Fungus	46	9	20%
Dexter	61	12	20%
HiddenVNC bot	60	12	20%
Predator	30	9	30%
Prostostealer	70	14	20%
Conti ransomware	93	14	15%
Babuk ransomware	35	11	30%
RedPetya	15	9	60%
ransomware	9	9	100%

参考文献

攻读学位期间发表论文与研究成果清单